Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020, 7(5):130-133

Research Article ISSN: 2394 - 658X

Impact of Object-Oriented Design on Software Testing

Naga Sai Krishna Mohan Pitchikala

Department of Computer Science
University of Texas at Dallas, Dallas, TX.
Nxp180022@utdallas.edu

ABSTRACT
Object-Oriented Software Development (OOSD) is a technique that involves building software by organizing the system around objects which represent real-world entities. These objects interact with each other to perform tasks, making the system more modular, reusable, and maintainable. OOSD's flexibility and scalability have made it a popular choice among developers. Object-Oriented Design (OOD), a crucial aspect of OOSD, introduces challenges in software testing due to concepts like encapsulation, inheritance, and polymorphism. The internal state of objects is concealed by encapsulation, making it challenging for testers to verify what's happening inside an object without using public methods. While it is possible for subclasses to inherit behaviors from parent classes through inheritance, testing is challenging due to the potential impact of changes in parent class boundaries on all child classes. With polymorphism, the complexity of testing requires more scenarios as objects can be created in various forms and methods are dynamically bound at runtime. Traditional testing methods often struggle with these concepts, so we need specialized approaches to handle them. This paper discusses solutions like scenario-based testing, incremental testing for inheritance, and the use of testing frameworks like Junit to handle these problems. These approaches help address the challenges of testing object-oriented systems, ensuring that software quality is maintained via testing despite the complexities introduced by OOD.

Keywords: Object-Oriented Software Development (OOSD), Object-Oriented Design (OOD)

INTRODUCTION

Object Oriented Software Development:

Object-Oriented Software Development (OOSD) is a methodology used for designing and developing software systems based on the principles of Object-Oriented Programming (OOP). It focuses on organizing software around objects (which represents real-world entities) or concepts that have both data (attributes) and behaviors (methods). This method suggests that software should be designed and developed to replicate our interactions with physical objects to make it more adaptable, manageable and scalable.

Key Concepts of Object-Oriented Software Development:

1. Objects: Objects are the fundamental blocks of object-oriented software design (OOSD). They represent entities from the real world by combining data (also known as attributes or properties) and actions (often called methods or functions). For example, in a banking system the customer account would be an object with data being the customer's name and account balance and actions being depositing or withdrawing money.

2. Classes: Classes are the blueprint for creating objects. They define the structure and behaviors that the objects created from the class will inherit. For example, a class “Car” will define properties like color and speed and methods like accelerate or brake. Objects created from this car will inherit those properties. Like Car-A and Car-B are the instances of the class Car which will inherit those properties.

3. Key Principles: As stated in the earlier part OOSD is based on the principles of Object-Oriented Programming. There are 4 such principals [1, 2]

a. Abstraction: Abstraction is a process where the implementation details of an object are hidden. This simplifies the user interaction with objects by focusing on what the object does, rather than how it does it.

b. Encapsulation: Encapsulation is a process of bundling the data and actions inside the object while keeping the internal details hidden. This approach restricts the access to the object inner methods and protects it from being changed in unexpected ways which improves data security and modularity.
c. Inheritance: Inheritance allows a new class (subclass) to inherit the characteristics (data and actions) of an existing class (superclass). This improves the code reusability.
d. Polymorphism: Polymorphism allows objects from different classes to be used in the same way because they share a common superclass. Even though these objects may act differently (because they are from different classes), they can still be accessed by the same set of actions.
The Process of Object-Oriented Software Development:
1. Object-Oriented Analysis (OOA): This is the first phase in the OOSD. In this phase, developers gather all the requirements to identify the key objects or entities in the system. Then they focus on understanding the problem domain and defining what the system should to solve the problem usually this is done using models like class diagrams and use case diagrams to identify the objects.
2. Object-Oriented Design (OOD): After identifying the objects in analysis phase, the next step is to design how the interaction between these objects within the system. This involves defining classes, relationships and how they collaborate to fulfill the system's requirements.
3. Object-Oriented Programming (OOP): After the design phase it the implementation phase. In this phase the design is implemented through code using an object-oriented programming language like Java, C++, or Python. Developers define and write the classes to create objects and implement the system's functionality based on the design.

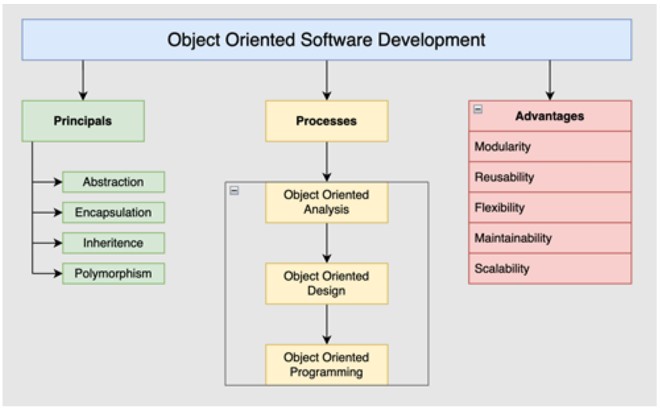

Fig. 1. *Object-Oriented Software Development Methodology*

Benefits of Object-Oriented Software Development:
• **Modularity:** The objects combine both data and actions into one unit, and they can be created and worked on separately from other parts of the system. This helps in breaking down the complex system into smaller and more manageable pieces, making it easier to understand and develop.
• **Reusability:** By the means of inheritance OOSD provides a way reuse the code in different parts of the system or even in future projects. By this means the DRY principle (Don't repeat yourself) of coding is also implemented which saves time and effort
• **Flexibility:** Creating new objects or extending existing ones allows new features and functionalities to be brought in which, in turn, helps respond to new requirements. Such flexibility allows the developers to extend the systems instead of the need to rewrite them entirely.
• **Maintainability:** Due to encapsulation, all the data and actions of the objects are hidden from the outside, thus making it easy to manage changes. When changes are needed, it is only necessary to change a few objects or classes and not the whole system thus making it easier to maintain.
• **Scalability:** As systems become larger or transform in any way, it is easy to add new objects or to extend existing ones. It is also possible to build upon the existing architecture by defining new classes which derive from other classes or use existing ones without heavy rearchitecting.
Now that we have a clear understanding of Object-Oriented Software Development (OOSD) and ways in which it enhances the software development process to a great extent, it is important to explore the drawbacks that OOSD introduced into the software testing. In the following sections of the paper, we will analyze these problems in the detail. To begin, we will start with examining the literature available on this issue to understand how OOSD affects testing. After that we will identify specific testing issues that arise from key object-oriented concepts like encapsulation, inheritance, and polymorphism and finally, we will discuss effective ways and approaches to handle

these challenges such as scenario-based testing, incremental testing and the use of modern testing frameworks. By understanding these approaches, we aim to increase the software quality even in complex object-oriented systems.

LITERATURE REVIEW

The Problematic of Testing Object-Oriented Software [4]:

The paper focuses on the issues of object-oriented software testing methodologies, especially due to concepts such as encapsulation, inheritance and polymorphism. The authors make it clear that methods of testing which were developed to deal with procedural languages cannot be used in oops and they must be modified. One of the solutions offered is reusing the inheritance of behaviors from class hierarchies to cut down the amount of testing done. They also caution that this practice could complicate the development of tests.

Testing Object-Oriented Systems: Models, Patterns, and Tools [5]:

This book serves as a comprehensive guide to testing object-oriented systems. It details how principles of OOP, such as polymorphism and inheritance, introduce complexities into traditional testing approaches and explains the need for new techniques, known as design patterns, to address the testing challenges in object-oriented systems. The author introduces various test design patterns, such as factory patterns for object creation during tests and observer patterns to validate object states indirectly, offering practical solutions to OOP-specific testing challenges. These patterns make the process of testing object-oriented systems more systematic and effective by providing ways to handle the complexity introduced by OOP features.

Class Testing and the Unified Testing Framework [6]:

In this book the author focuses on testing the individual classes in object-oriented systems by looking at the "state" of objects and how their behavior changes over time, rather than just testing the specific methods. This approach is useful for systems that are quite complex as there are units or classes which interact with one another and thus change the way the entire system works. This approach emphasizes on such important aspects as state-based testing where tests depend on various states of the object under test and also calls for utilizing scenarios to help cope with polymorphism making sure that proper test of interactions between objects of different classes is performed. This technique is especially important when it comes to dealing with the problems associated with the class interaction of an object-oriented programming language.

CHALLENGES IN TESTING OBJECT-ORIENTED SYSTEMS

While Object-Oriented Design offered many benefits with the introduction of the concepts like encapsulation, interference and polymorphism, it also introduced some challenges to the aspect of software testing which are discussed below

• **Encapsulation:** Encapsulation hides the object’s internal workings and only allows accessing through its public methods. While this provides better security and stability, it makes the testing of such objects rather problematic as the testers are unable to see or change the private data contained within the object itself. Rather, they have to use the public methods which also has the limitation on how deep the internal state of the object can be examined.

• **Inheritance:** Inheritance helps subclasses to inherit the attributes present in the parent class thus increasing code reusability. However, this makes testing hard since any modification made in one parent class can affect all the subclasses. In other words, methods that work perfectly in the parent class may behave differently in some of its subclasses meaning both the parent and child classes need to be carefully tested.

• **Polymorphism**: Polymorphism allows subclasses to override methods where the specific method that gets called through invocation is decided at runtime. This approach makes the testing challenging because it will be harder to predict which method will be called in different scenarios. As the number of child classes increase, so do the number of test cases, hence making it hard to test every method combination possible.

Additional Challenges in OOD Testing

• Since OOD involves complex interactions between objects, testing requires creating real-world scenarios where objects interact. This type of testing is more detailed and takes more time than traditional testing methods.

• Object-oriented systems rely heavily on message passing between objects, which makes it harder to predict and test how objects will communicate with each other. As a result, more advanced testing techniques are needed to ensure the system behaves as expected under various conditions.

OVERCOMING THE CHALLENGES OF TESTING OBJECT-ORIENTED SYSTEMS

These approaches provide structured solutions to overcome the testing challenges introduced by object-oriented systems, ensuring robust and reliable software quality.

Scenario-Based Testing

Scenario-based testing is a popular method for dealing with the challenges associated with testing polymorphic systems. By using this approach, it is possible to create realistic situations in which objects from different classes interact, ensuring that objects behave as expected, particularly when subclass methods are overridden. It helps in verifying that the appropriate method is invoked during polymorphic interactions, making it simpler to evaluate dynamic behaviors in controlled conditions.

Incremental Testing for Inheritance

Subclass behavior can be validated through incremental testing in inheritance-based systems. Rather than repeating all inherited methods, this technique concentrates on testing the new or modified functionality in the subclass. This reduces redundant testing while ensuring thorough verification. Additionally, regression testing ensures that changes in the parent class do not negatively impact the subclasses.

Testing Frameworks

Frameworks such as JUnit (for Java) save developers from the complexities of testing object-oriented software. These frameworks allow developers to write unit tests that target public interfaces, indirectly testing encapsulated behaviors. They also support testing different instances of polymorphism by using mock objects to simulate the interactions between class instances making testing more efficient while adhering to encapsulation principles.

Class Hierarchy Testing

Class Hierarchy Testing is one of the most important testing techniques in OOD. It is the practice of testing the whole class hierarchy, right from abstract base class to multiple derived classes. It ensures the correct implementation and extension of behaviors that are common to its subclasses. For each and all levels of hierarchy, test cases are created to confirm that behavior at that level in the hierarchy is as it should be, and that subclasses comply with the obligations of the super classes.

CONCLUSION

The techniques in object-oriented design have been the building blocks of present-day software development. However, these principles make existing testing methods more challenging, particularly with features such as encapsulation, inheritance, and polymorphism. This has led to the development of new approaches for testing object-oriented software, such as scenario-based testing, incremental testing for inheritance, mock objects, and modern testing frameworks. These methods help ensure that OOD systems meet performance requirements, maintain high quality, and are easier to manage. To fully benefit from OOD while maintaining high-quality software, developers need to adopt testing strategies that handle the complexity of OOD. As object-oriented programming becomes more widely used, adapting testing processes to fit OOD is crucial for sustaining software quality in the future.

REFERENCES

[1]. https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/
[2]. https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727
[3]. Object-Oriented and Classical Software Engineering Eighth Edition by Stephen R. Schach
[4]. www.witpress.com/Secure/elibrary/papers/SQM94/SQM94029FU2.pdf
[5]. Testing Object-Oriented Systems: Models, Patterns, and Tools by Binder, R. (2000)
[6]. Schnizler, Moritz & Lichter, Horst. (2000). Test Automation for Object-Oriented Frameworks.
[7]. https://qatestlab.com/resources/knowledge-center/scenario-based-testing/
[8]. https://www.veracode.com/blog/secure-development/static-testing-vs-dynamic-testing